Master's Thesis

Persistent Database Buffer Caching and Logging with Slotted Page Structure

Jihye Seo

Department of Computer Science and Engineering

Graduate School of UNIST

2018

Persistent Database Buffer Caching and Logging with Slotted Page Structure

Jihye Seo

Department of Computer Science and Engineering

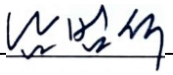
Graduate School of UNIST

Persistent Database Buffer Caching and Logging with Slotted Page Structure

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Jihye Seo

12. 14. 2017
Approved by



Advisor
Beomseok Nam

Persistent Database Buffer Caching and Logging with Slotted Page Structure

Jihye Seo

This certifies that the thesis of Jihye Seo is approved.

12. 14. 2017

signature

Advisor: Beomseok Nam

signature

Sam H. Noh: Thesis Committee Member #1

signature

Woongki Baek: Thesis Committee Member #2

three signatures total in case of masters

Abstract

Emerging byte-addressable persistent memory (PM) will be effective to improve the performance of computer system by reducing the redundant write operations. Traditional database management system uses recovery techniques to prevent data loss. The techniques copy the entire page into the block device storage several times for one insertion, so the amount of I/O is not negligible.

In this work, we consider PM as main memory. Then, the durability of data in the buffer cache is ensured. To guarantee consistency, we exploit slotted page structure which is commonly used in database systems. We revisit that the slot header, which stores the metadata of the page in the slotted page structure, can act like a commit mark in the persistent database buffer cache.

We then present two novel database management schemes using persistent buffer cache and slotted page. In-place commit scheme updates the page atomically using hardware transactional memory. It doesn't make any other copies and has optimal performance. Slot header logging scheme is needed for the case of updating pages more than one. Unlike the existing logging technique, slot header logging reduces the write operations by logging only commit mark.

We implemented these schemes in SQLite and evaluate the performance compared with NVWAL, which is the state-of-the-art scheme. Our experiments show that in-place commit scheme needs only 3 cache line flush instructions for one insertion and slot header logging scheme reduces logging overhead at least 1/4.

Contents

List of Figures

List of Tables

I. Introduction

Persistent memory (PM) such as phase-change memory (PCM) and spin-transfer torque magnetic RAM (STT-MRAM) is expected to be the new memory for improving the performance of computer system [1-3]. PM is considered as fast and byte-addressable like DRAM and persistent like block device storage. Recent researchers have tried to redesign the legacy system to exploit the properties of PM [2-6].

Database management system is one area that requires fast, persistent and byte-addressable properties. Traditional DBMS should frequently access slow block device storage to prevent data loss. When a transaction inserts a data, an entire page where the data will be inserted is copied into the volatile buffer cache from storage. To guarantee durability, the dirty page must be written to persistent storage after updating the page in the volatile buffer cache. Before performing write operations to copy the page into persistent storage, recovery techniques such as journaling and logging keep the original page for avoiding partial write because the write operation is not atomic. Thus, DBMS makes significant write traffics for inserting a single record.

Another problem is that file system layer also relies on journaling technique. Hence, when we insert a record into the database, the updated metadata of database file and database journal file are written into the file system journal for ensuring consistency. This journaling of journal problem is known to amplify the amount of I/O in flash memory devices [7-9].

Several previous studies used PM as fast secondary storage for resolving slow performance of DBMS and file system [2, 5]. Oh et al. proposed new optimization logging strategy, called per-page logging (PPL) for mobile database management [5]. They used PCM as a fine-grained logging device to minimize write amplification in flash memory devices. Kim et al. also modified logging techniques leveraging the properties of PM [2]. While PPL creates a new log structure, NVWAL uses a differential logging method that only logs the changed part, not the entire page. Both techniques reduce logging overhead and avoid Journaling of journal, but they have page duplication overhead to the volatile buffer cache and database file in storage.

In this work, we consider PM as main memory. Then, durability is guaranteed even if data is not flushed to block device storage because database buffer cache is persistent. This PM-only system can eliminate redundant copies of an entire page and reduce the number of write operations. However, there is still the possibility of data loss against system failures because today's processors can reorder the memory write operations. Previous works used memory fence and cache line flush instructions, but there are expensive instructions, so we must use them carefully.

To guarantee consistency and atomicity for persistent database buffer cache, we revisit slotted page structure which is commonly used in DBMS such as SQLite, PostgreSQL, and InnoDB for variable length records [10]. In slotted page structure, metadata of the page and records are divided at the beginning and end of the page. Since records are inserted in an append-only manner, the slotted page structure manages the order and validation of the records by the metadata part, which is defined as the slot header, through the record's offset. That is, if the offset of a record is included in the slot header, the corresponding record becomes valid. Combining this feature with the byte-addressability of the PM, we decided to use the slot header as a commit mark.

Using persistent slotted page, we propose two novel database management schemes, in-place commit, and slot header logging scheme. The goal of these two schemes eliminates page duplication and keeps an only single copy. In-place commit scheme does not use journaling, logging, or copy-on-write techniques, and it just uses a single atomic write operation for transaction commit. This scheme reduces significant redundant write operations. Slot header logging scheme ensure the consistency and atomicity using logging method and it reduces logging overhead by copying only slot header, not the entire page.

For in-place commit scheme which exhibits optimal performance, we take the method that directly updates the original page, not making page duplication. The problem is that it is difficult to update slot header atomically because the atomic write of PM is expected to 8 bytes [1, 6], but slot header is larger than 8 bytes. To solve the problem, among the previous studies that increased the granularity of PM, we use the hardware transactional memory proposed by Dulloor et al [11]. It can flush the cache line sized data atomically, so if the slot header is smaller than a cache line, the slotted page can be updated atomically.

If hardware transactional memory is not supported or a transaction updates more than one pages, we can't use in-place commit scheme. In this case, the slot header logging scheme is used because the logging technique is inevitable. Slot header logging scheme writes only the slot header, which is same as commit mark of page update, in the persistent slot header log and uses early checkpointing to eliminate the overhead of searching the slot headers stored in the log for the next transactions.

Both schemes were implemented in SQLite, which is most widely used as mobile database management system. We created two versions, Failure-Atomic Slot Header logging (FASH) and Failure-Atomic Slot header logging with in-place commiT (FAST). FASH uses only slot header logging schemes and FAST uses both in-place commit and slot header logging schemes. We used Quartz [12] for emulating PM latency and compared our proposed database management schemes with NVWAL, which is the state-of-art work [2]. FASH reduces database logging overhead to 1/4 and

FAST improves query response time by up to 33 % compared to NVWAL.

In the rest of this paper, we explain background in Section 2. The design of failure-atomic slotted paging schemes and two implementation versions in SQLite is described in Section 3 and 4. Then, we evaluate the performance of FASH, FAST and NVWAL in Section 5. At the end, we conclude this paper in Section 6.

II. Background

In this section, we introduce background studies for easily understanding our proposed work. At first, we explain the legacy recovery techniques of database management systems - journaling and write-ahead logging. Next, we review the properties of persistent memory and related works which use persistent memory to improve file system or database system performance. Lastly, we describe the slotted page structure which is commonly used in database management systems. The slotted page can be a solution to consistency and atomicity problem in persistent memory.

2.1 Recovery Techniques in Database System

Database management system should take careful consideration of atomicity, consistency, isolation, and durability. There is considerable overhead to satisfy all four conditions. When we want to insert data, DBMS first copies the page to the fast DRAM buffer cache. It then updates the page in the volatile buffer cache and reflects it in persistent storage for durability. If a system crash occurs while flushing updated pages to storage, the database file may be corrupted. For guaranteeing atomicity and consistency, DBMS needs recovery techniques. The most commonly used techniques for recovery techniques are journaling and write-ahead logging [2, 5, 8].

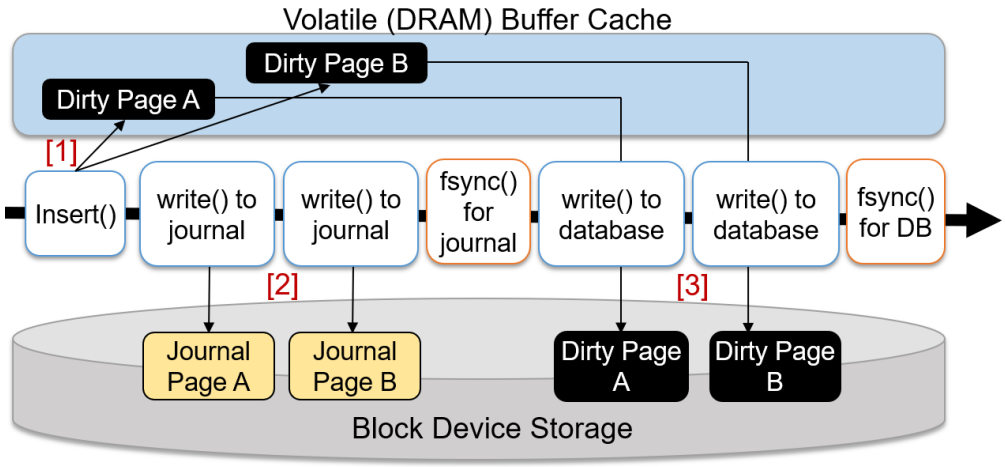


Figure 1. Journaling

The process of journaling done in SQLite is shown in figure 1 [3, 7]. It has journal file in block device storage to store original pages during page update. When we insert data into the page A and B in figure 1, (1) DBMS updates pages in the volatile buffer cache. (2) It then copies original pages from the database file into the journal file. After synchronizing the journal file, (3) it writes dirty pages from the volatile buffer cache to the database file. At the end, it synchronizes the database file and empties the journal file. If a system crash occurs during writing dirty pages, database file can be

recovered by copying the journal file to the database file.

This journaling process creates a single copy in volatile memory and writes two copies in persistent storage. Hence, it doubles the amount of I/O in the database layer. Furthermore, EXT4 file system layer amplifies the write traffics due to file system journaling [7, 8] because the journaling method in the file system copies the metadata blocks of both database and journal files.

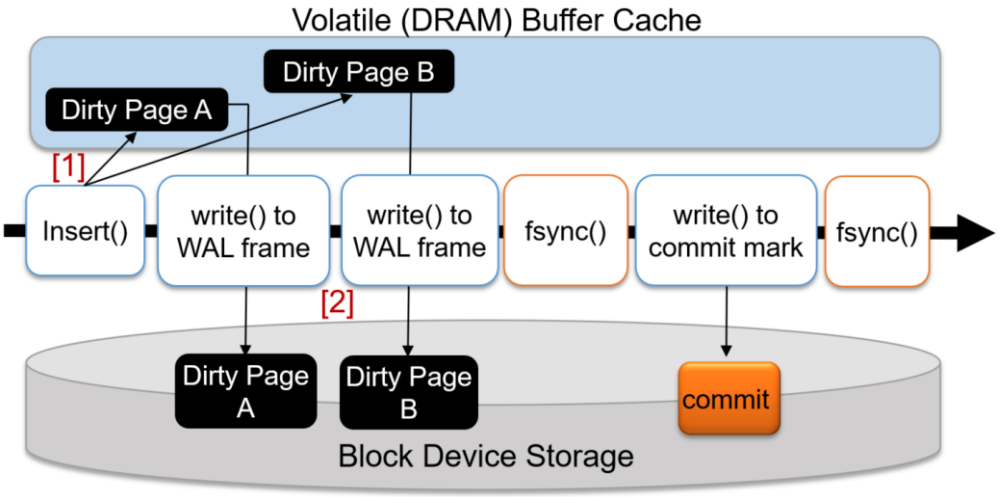


Figure 2. Write-ahead Logging

A write-ahead logging technique has been introduced to address the disadvantage of journaling that performs write operations both journal and database files per insertion. When data is inserted, (1) DBMS updates page A and B in the volatile buffer cache like journaling. And then, (2) it writes the updated page from buffer cache to the WAL file in the storage. After synchronizing WAL file, it writes commit mark to the WAL file.

If a system failure occurs when writing the updated pages to the WAL file, DBMS has original page A and B in the database file, so it ignores the logged pages without the commit mark. If a system failure occurs after the commit mark, DBMS copies the updated pages from the WAL file into the database file. When the WAL file is full, the pages in the WAL file is copied to the database file. Since the write operations for the WAL file are only performed when checkpointing occurs, WAL technique performs faster than journaling. It helps mitigate the journaling of journal problem [7]. However, both techniques have the disadvantage of creating redundant copies to volatile buffer cache and block device storage and writing the entire page even if we want to modify part of a page.

In this work, we try to keep a single copy and avoid redundant write operations exploiting PM. For persistent buffer cache, journal or WAL file doesn't need to be in slow storage and we can't need to copy the entire page in journal or log. The key technical challenge is to update directly on PM while

assuring that recovery is possible in case of system crashes. This is not simple because modifying the existing data by overwriting can result in inconsistent states if a system crashes at inopportune times. This paper proposes an efficient PM-exploiting scheme that targets slotted-paging.

2.2 Persistent Memory and Related Work

Emerging persistent memory technologies such as STT-MRAM and PCM are expected to change the landscape of memory and storage systems. This next generation memory has the advantages of both DRAM and block device storage [13]. It is non-volatile and large capacity like storage and fast and byte-addressable like a DRAM. Then, we can replace block device storage or main memory with persistent memory to improve performance.

When we use persistent memory, it is important to use cache line flush and memory fence instructions carefully. Today's processor can change the order of memory write operations. Hence, if a system failure occurs when the write operation that validates the data is written first before the write operation for the data, the erroneous information is persistently preserved. Such rearrangement of the order of write operations hurts consistency. Thus, many studies have used cache line flush and memory fence instructions. However, since these instructions are expensive, reducing the number of uses is also one of the considerations.

PM gives new challenges in redesigning file systems and DBMS by leveraging the durability and high performance [2-4, 9, 14]. If we add PM to the existing architecture including both DRAM and storage, PM is primarily used to store the part that causes performance bottlenecks. In the file system layer, the cause of performance degradation is journaling method because it doubles the number of write operations. Lee at al. introduced UBJ which puts the buffer cache in PM and combines the journaling into the buffer cache [3]. UBJ removes filesystem journaling, but it still requires a separate recovery method to ensure consistency of database transactions. Kim et al. proposed delta journaling which leverages byte-addressability of PM [4]. It saves file system journal to PM and stores only the differences, not the entire page, to reduce journaling overhead.

For database layer, database journal or log files are stored in persistent memory to eliminate page duplication and journaling of journal problem. Oh et al. proposed SQLite/PPL which is a new database logging technique [5]. SQLite/PPL stores per-page log in PM and logs the update information per each page. It attempts to reduce the number of write operations using the small log format they proposed instead of page writes. Kim et al. proposed a novel logging method, NVWAL which stores write-ahead log in PM [2]. NVWAL uses differential logging to avoid page duplications. However, both methods fail to remove redundant copies of DRAM by using a volatile buffer cache. In

addition, NVWAL copies all dirty portions of data pages to the log in PM, while our schemes update the dirty record directly.

If we consider PM as the main memory, data in memory will be persistent and access to storage will not be needed. As well as, if a transaction is guaranteed to be atomic, then journaling is no longer necessary. It is important how to ensure consistency and atomicity for all updates by less using expensive cache line flush and memory fence instructions. We propose a novel database management scheme based on this PM-only system.

In the experimental section, we compared our proposed method based on PM only system with NVWAL based on the hybrid memory system. As a result, the PM only system performs better than the hybrid memory system despite the higher PM latency than DRAM.

2.3 Slotted Page Structure

Traditional disk-based database systems use various file formats to store database tables. One of the file formats is B-tree file format. B-tree file stores variable-length records in the leaf pages and makes easy to find records. Among the page structures that make up the file, slotted page structure is the most widely used database page format for variable-length records in a fixed-sized block [10]. It is used in databases such as SQLite, PostgreSQL, InnoDB and so on.

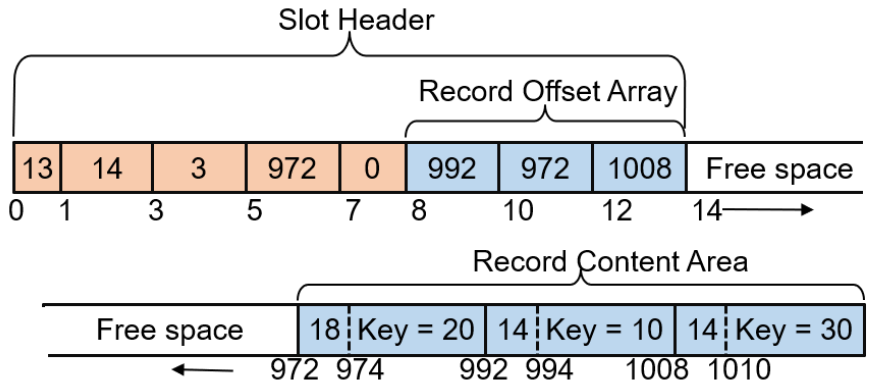


Figure 3. Slotted Page Structure

Figure 3 shows the layout of slotted page structure. At the front of the page, slot header stores the status of the page including record offset array. Records are written from the end of the page in an append-only manner. The area is called record content area. Each record consists of the size of the record, key, and value. In figure 3, each record is represented only by the size and key. The order of records is managed by record offset array according to the key of the record. According to the record offset array in the slot header, the offset 992 of the record with the smallest key 10 is stored first.

Subsequently, record offset 972 with key 20 and record offset 1008 with key 30 are stored.

Offset	Size	Description
0	1	Flag
1	2	Start of first free block
3	2	The number of records
5	2	Start of record content area
7	1	Number of fragmented free bytes
8	2	Offset of the smallest record
10	2	Offset of the 2nd smallest record
…	…	…

Record offset array

Table 1. Description of Slot Header in Slotted Page Structure

Table 1 indicates the descriptions of each offset. At offset 0, page flag means that the page is leaf or interior. Start of the first free block at offset 1 refers to the first free space in the record content area when records are deleted. The remaining empty spaces are linked as a list by storing the offset of the next free space in the previous free space. The number of fragmented free bytes at offset 7 is the total bytes of the small space that cannot fit into the record. After metadata, there is a record offset array that stores the records' offset in smallest order in 2 bytes.

III. Failure-atomic Slotted Paging

In this work, we target persistent database buffer cache by replacing DRAM with the PM. We then can write and read the partial part of the page. However, if a system crashes while a transaction is making changes to the page in persistent memory, the buffer cache may have partially written inconsistent data. To guarantee consistency, the update must be invisible until the transaction commits. Slotted page structure is suitable to take the advertisement of the persistent buffer cache. In this section, we revisit the property of slotted page structure and propose two novel database management schemes.

3.1 Persistent Slotted Page Structure

In block device storage, copy and update query occur in 4K and 8K units even if a user modifies a small part of a page. Leveraging byte-addressability of PM, we can update only what we want to modify. To ensure consistency and atomicity of all partial modifications in a transaction, the dirty records should be kept invalid and should not change the committed record until the transaction commits. Then, all modifications should be validated with a single write operation, which is a commit mark. Slotted page structure can be updated while satisfying these conditions.

Slotted page structure manages records in a page with a small sized slot header. The append-only manner used in the record content area makes it possible to update a record directly because it does not hurt existing committed records. The updated record becomes valid when slot header has the record's offset. Only after updating the slot header, the record becomes visible. In other words, slot header can be used like a commit mark. The features of slotted page structure can guarantee consistency and atomicity of transaction with a small number or write operations in the persistent buffer cache.

3.2 In-place Commit Scheme

It is important to atomically update the slot header because the slot header is used as commit mark. For the purpose, we introduce in-place commit scheme. This scheme achieves fast performance by minimizing redundant write operations because it uses only one page and validates all modifications in the page with a single failure-atomic write. The problem is that failure-atomic write in PM is expected to be supported at 8 bytes [1], but the minimum size of slot header is 8 bytes.

Hardware transactional memory such as the Intel's Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) is one solution to the problem. This instruction helps to write coarse-grained data (cache line seized data) atomically via hardware support [15, 16]. In-place update

scheme uses RTM because we can define fallback execution path against system failure. In RTM, XBEGIN, XEND, and XABORT instructions are provided. XBEGIN indicates the start of hardware transaction and XEND indicates the end of the transaction. XABORT is used when the transaction cannot be successfully committed [11, 16]. Hardware transaction stores dirty cache line written after XBEGIN in the write combining store buffer. It is ensured that the dirty cache line stored in the write combining store buffer is not visible outside the transaction and is not flushed to memory until XEND instruction successfully completed.

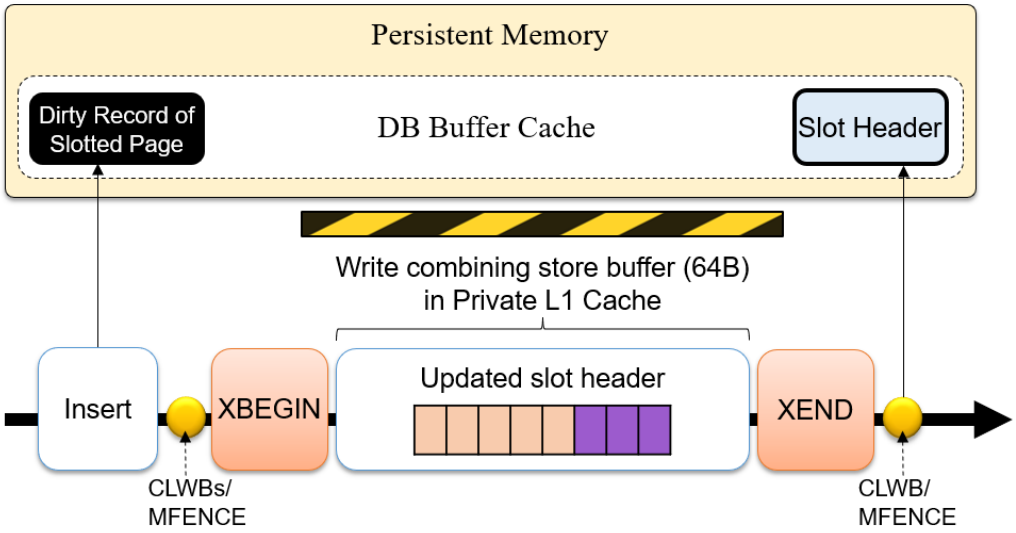


Figure 4. Insertion Process of In-place Commit Scheme

If a system crashes before XEND instruction, the dirty data in write combining store buffer is lost and it doesn't affect the consistency of committed data in persistent memory. Then, XABORT instruction executes fallback handler which iterates until the transaction succeeded. If transaction failure continues, the fallback handler uses our other proposed scheme, slot header logging instead of in-place update scheme.

Figure 4 shows the timeline of inserting a record. When a transaction inserts a record into a slotted page, it writes the record directly along with its length at the front of record content area. In this step, it does not need to atomically write the record. It is because if a system failure occurs while writing the record, the page status for other transactions will look the same as it was before the transaction started. Before updating the slot header, it must be ensured that the record is written to the persistent buffer cache. So, we call cache line flush instructions and memory barrier instructions for the record.

Secondly, the transaction updates the slot header which should be written atomically. In the slot header, both metadata, which includes the number of records and first bytes of record content area,

and the record offset array should be modified to validate the inserted record. If the size of slot header is smaller than cache line size, RTM can ensure the failure-atomicity for cache line write. So, before writing the updated slot header, we call XBEGIN instruction. After that, the updated slot header is stored in the write combining store buffer and we call XEND instruction when the writing is completed. Since the slot header is in the private L1 cache after RTM, a cache line flush instruction is needed to atomically write the slot header to PM. This failure-atomic slot header writes same as an in-place commit mark guarantees consistency and durability for the insertion transaction.

Let's look at the update and delete methods of in-place commit scheme. When a transaction updates a record, it should not overwrite the record because uncommitted value should not be considered valid. So, the transaction inserts the updated record as a new record and replaces the original record offset with the new record offset when atomically updating the slot header. After this commit mark, the original record becomes invalid and the new record becomes valid.

When a transaction deletes a record, it just modifies the slot header in the persistent slotted page. Since the record is invalid by deleting its offset and decreasing the number of records, we do not need to modify record content area. Failure-atomic update of slot header is ensured by RTM transaction.

A problem of update and deletion is that they make the empty hole in the slotted page. These empty holes in the record content area are connected by a free list. To make the record free space, we must overwrite the offset of the next free space in the record. Since the committed record should not be overwritten during the execution of the transaction, the free list is built after the transaction is committed. If a system crashes during connecting the free list, we can re-establish free list by searching the valid records in the record offset array. Another problem with the free space is defragmentation. We handle the defragmentation problem in Section 4.4.

In-place commit scheme keeps only a single page and shows optimal performance when a transaction updates a single page, but there exist limitations. One is hardware devices that support RTM are limited. Another is that the size of slot header is no larger than the hardware limit. If the slot header is larger than a cache line, we cannot use the in-place commit scheme. The other is that RTM cannot guarantee atomicity for multiple writes transaction. Write combining buffer cache includes multiple consecutive small 8 bytes writes, so separate slot headers cannot be updated atomically. Hence, we propose the other recovery scheme.

3.3 Slot Header Logging Scheme

The B-tree format, which consists of a slotted page structure, causes a page splitting that modifies multiple pages when the page is full. Furthermore, common enterprise database systems insert more

than one records in a single transaction. However, the in-place commit scheme is not suitable for multiple writes in a single transaction. To guarantee failure-atomicity, we propose the other scheme, slot header logging scheme.

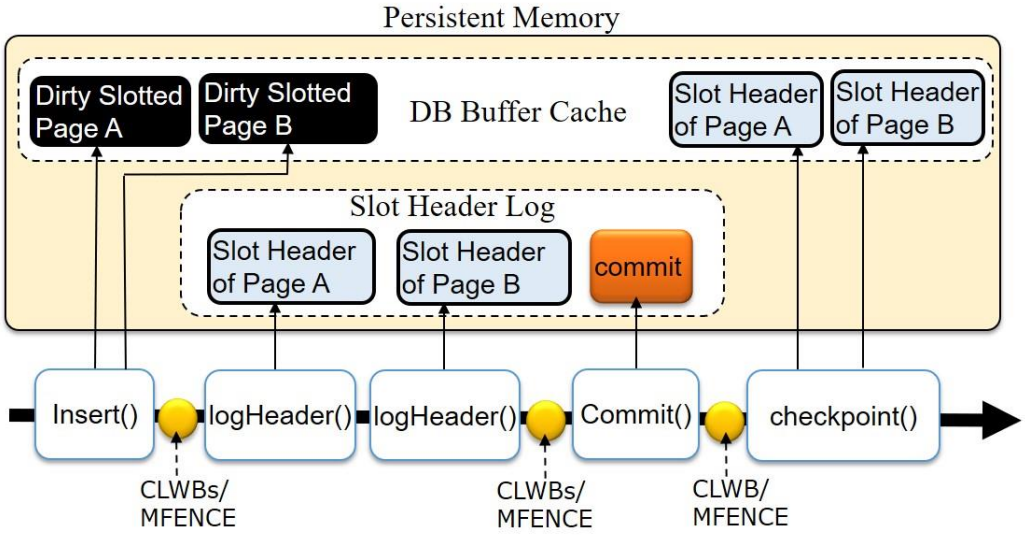


Figure 5. Insertion Process of Slot Header Logging Scheme

Slot header logging scheme has the persistent buffer caching, slot header logging and an extra commit mark that determines whether the slot header in the log is committed or not. There are two differences between slot header logging scheme and existing write-ahead logging technique. First is that slot header logging scheme stores only the slot header of pages, not the entire page. Insertion a record into the record content area does not affect the consistency of the page, so we do not need to duplicate the record content area. We write just the slot header in the slot header log in PM because slot header does the same role with a per-page commit mark. Since the size of the slot header is considerably smaller than the size of the page, we can maintain small slot header log and minimize the memory write operations.

Second is that slot header logging scheme checkpoints slot header logs as soon as a transaction is committed. Since copying logs to block device storage is slow, write-ahead logging delays checkpointing until the log is full to prevent performance degradation. In persistent database buffer cache, copying data to PM is relatively fast, so using eager checkpointing can reduce the overhead of searching the slot header logs every transaction. Hence, slot header logging also reduces the memory read operations.

Figure 5 illustrates the insertion process of slot header logging scheme. When a transaction wants to insert records into page A and page B, at first, records are in-place written in the record content area of

each page. The new records do not hurt the consistency of page A and B. Before logging the slot headers, we should ensure that the records are flushed to persistent buffer cache, so we call cache line instructions and memory barrier instructions like the in-place commit scheme. If a system crashes before logging, we can ignore dirty records because they are invisible.

After writing records, slot header logging scheme writes the slot header of page A and B into the separate slot header log. When modifying each page, it writes page's slot header to the log but does not flush immediately. The cache line flush instructions are called for the logs only when modifications in the transaction is finished. The order of flushing the slot headers in the log is not important. This is because the log entries are all meaningless unless we put a commit mark in the log. This follows the approach taken in NVWAL [2]. Hence, cache line flush instructions for all logs are bound to a single memory barrier instruction back and front. It can reduce the overhead of memory barrier instructions. This method has the other advantage that if a transaction inserts records multiple times on a single page, it can flush only the last modified slot header instead of flushing the slot header for each insertion.

When logs are flushed completely, the commit mark for the transaction is written to the log and it is also flushed by cache line flush and memory barrier instructions. By marking the commit mark, the slot header logs, which are written before the commit mark, become valid. The recovery method is different before and after the commit mark is put. If a system crashes before we put the commit mark, the updated slot headers exist only in the slot header log, not in the actual pages, so we can ignore the slot header log. If a system crashes after we put the commit mark, we can just redo checkpointing.

After the commit mark is flushed to the slot header log, slot header logging scheme immediately execute checkpointing the slot header log to the persistent buffer cache. Unlike legacy checkpointing, our eager checkpointing does not require expensive write operation to block device storage and removes log checking overhead.

In the cases of update and deletion, the record processing is the same as the in-place commit scheme, but the slot header is stored in the separate slot header log instead of being written directly to the original page. Comprehensively, slot header logging scheme requires more memory write operations than in-place commit scheme, but it generates much less read and write traffic than traditional logging schemes.

IV. Implementation

We implement in-place commit scheme and slot header logging scheme in SQLite, which is embedded database system using B-tree file format. The B-tree structure consists of a leaf page with key and value and an interior page pointing to the leaf pages. In this section, we introduce two versions of implementation exploiting failure-atomic slotted paging schemes. We will discuss how to perform insertion, split, and defragmentation of B-tree.

4.1 FASH

First implementation version is Failure-Atomic Slot Header logging (FASH) which exploits the slot header logging scheme. Even if we insert a single record into the B-tree and a single leaf page is modified, FASH stores leaf page's slot header to persistent slot header log. It requires more memory write operations than the in-place commit scheme. Nevertheless, FASH does not require hardware transactional memory, and it is possible for one transaction to perform multiple insertions. As well as, since the size of slot header is not limited, many records can fit on a slotted page structure.

4.2 FAST

The second version is Failure-Atomic Slot header logging with in-place commiT (FAST) which uses both slot header logging scheme and in-place commit scheme. As the RTM used in the in-place commit scheme cannot guarantee failure-atomic write for more than one slot headers, FAST is used only when a transaction inserts a single record. When one record is inserted in B-tree and one leaf page is modified, FAST can use minimal I/O. However, there is a disadvantage that the number of records that can fit on the leaf page is small because the size of the slot header cannot exceed the cache line. The metadata size in the slot header of leaf page is 8 bytes and remaining size can be used for 2 bytes of record offsets. When a cache line is 64 bytes, leaf page can hold maximum (64-8)/2 = 28 records.

There is a case in which one or more pages are modified even if a transaction does a single record insertion. If a leaf page where a new record is written is full, then half of the records are split into a new leaf page. This new leaf page must be inserted into a parent page. Hence, it is necessary to modify two existing pages. In the case when multiple insertions are derived, FAST uses the slot header logging scheme. That is, modification of the interior pages is done by slot header logging scheme because it occurs only when the split occurs. Hence, the size of slot header in interior pages is not limited.

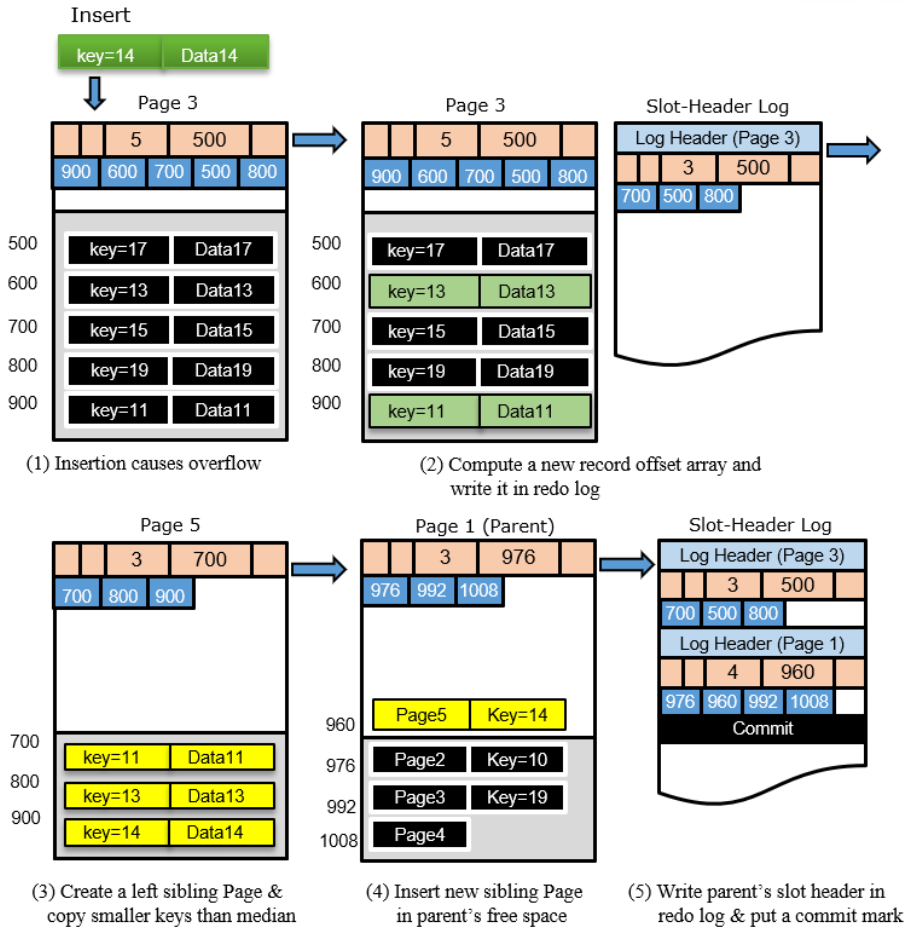


Figure 6. Split

4.3 Split

Tree structure splits records in a page when a record can no longer store on the page. Legacy SQLite redistributes the records to two sibling pages, so it modifies total four pages including overflowed page and parent page [8]. To reduce the I/O traffic, our split transaction causes the creation of one page and the modifications in the two pages - overflowed page and its parent page. It is similar to the method used in LS-MVBT [8]. Both FASH and FAST use the same method using slot header logging scheme. It is important to modulate the order of update to guarantee the consistency of all pages. Let's look at the process of split and checkpointing with an example.

Figure 6 indicates how split works. (1) When a record (key=14) is inserted into page 3, page 3 is split because there is not enough space for the record. (2) We will redistribute the half of smaller records from the page 3 to a new page and the other half of records remain on page 3. To invalidates the half of smaller records, we should modify the slot header of page 3. The slot header removes some record offsets and decreases the number of records. Then, it is written in the slot header log as redo

information. (3) In new sibling page 5, the records whose keys are smaller than the median key are copied. Since the new sibling page does not hurt the B-tree state, we do not need to store its slot header in the slot header log. After the in-place update of the new page, (4) the new page is connected to a parent page of the overflowed page. A record with page number and the largest key of page 5 is inserted in front of the record content area on the parent page. (5) Finally, the updated slot header of parent page is written in the slot header log and we put a commit mark which means the transaction is finished.

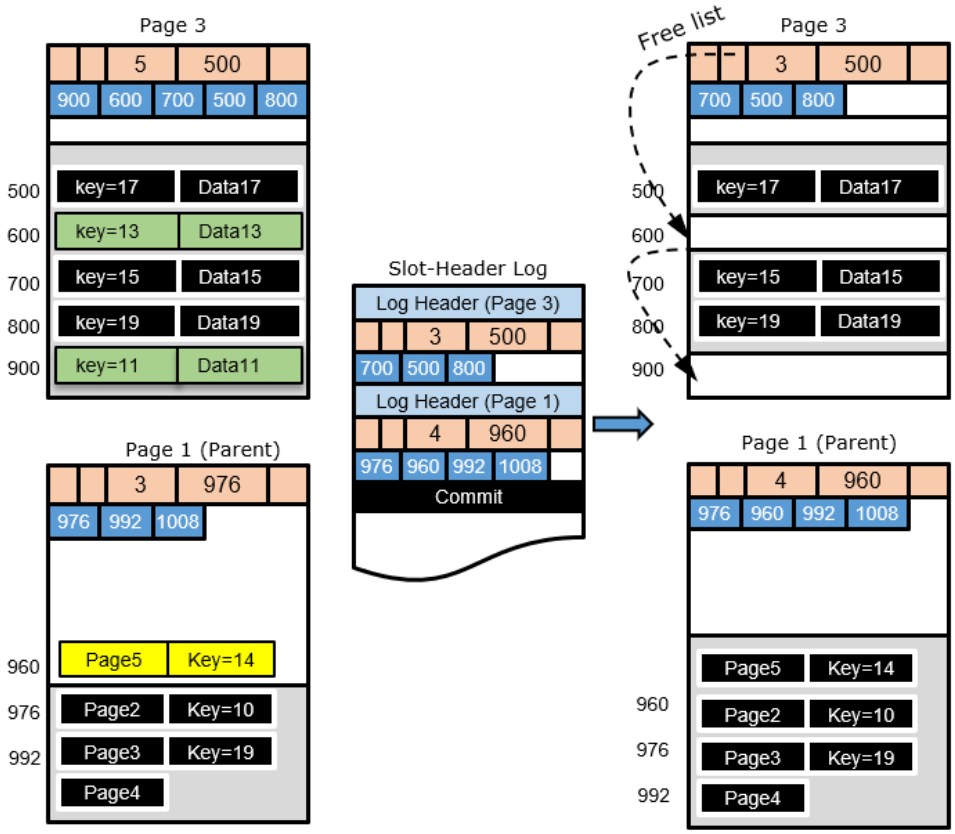


Figure 7. Checkpointing

When the transaction is committed, we perform checkpointing logs to the persistent buffer cache. It is shown in Figure 7. In the previous example, slot headers of page 3 and page 1 are stored in the redo log. In checkpointing phase, we copy the logs to original pages. It makes records with key 13 and 11 in page 3 invalid and a record with key 14 on page 1 valid. Now, we can overwrite the invalid records for the free list because other transactions find the records from the updated slot header. At offset 600, the size of free space and offset of next free space (900) are stored.

Recovery is simple. Before putting the commit mark, modifications in existing pages are invisible to other transactions and the slot headers written in the slot header log are invalid. We can ignore the

dirty records and the slot header logs without commit mark can be discarded. After the commit mark, we redo checkpointing the slot header logs with commit mark to the original page in the buffer cache.

4.4 Defragmentation

If the total size of free spaces is enough to store a record but they are fragmented, we perform defragmentation. In addition, when inserting a record into a page where page split occurs in the same transaction, there is not enough free space, so we need to perform defragmentation. For example, if we want to insert a new record on page 3 before we do checkpointing method in Figure 7, page 3 will have to make free space while preserving the committed records. For such case, defragmentation is needed. Legacy defragmentation is a method of collecting only valid records and accumulating them in order from the end of the page to eliminate free space. For persistent buffer cache, changing committed records can break consistency. Thus, we implement defragmentation using the copy-on-write method. Using the copy-on-write method, we can guarantee the consistency easily because all edits occur on the new page.

At first, we create a page and copy all valid records to the new page. After the writing is guaranteed to be complete, we replace the existing page with the new page. It means modifying the parent page that points to the existing page to point the new page. Since we cannot perform the in-place update to the record content area of the parent page, we add a new record and store the slot header in the log that invalidates the existing record.

Our defragmentation requires a lot of write operations, but it does not cause a significant impact on performance because it rarely happens. The performance impact of defragmentation is shown in the experimental section.

V. Evaluation

Our experimental environment is four Intel Xeon Haswell-EX E7-8860 v3 processors with 2.20GHz, 16x32KB instruction cache, 16x32KB data cache, 16x256KB L2 cache, and 40MB L3 cache and 256GB of DDR3 DRAM. The Intel Xeon Haswell processor supports Restricted Transactional Memory. Also, we set the scaling governor to performance to make the processors run at maximum frequency.

We implement our failure-atomic slotted paging in SQLite 3.8. We compared FASH and FAST against NVWAL [2], which is the state-of-the-logging scheme proposed by Kim et al. and leverages both PM and volatile DRAM.

	NVWAL	FASH	FAST
Single page update	Differential logging	Slot-header logging	In-place commit
Multiple pages update			Slot-header logging
Buffer cache	In DRAM	In PM	In PM
Log	In PM	In PM	In PM

Table 2. Properties of NVWAL, FASH, and FAST

Table 2 summarizes the differences NVWAL, FASH, and FAST. Unlike FASH and FAST, NVWAL places the buffer cache in DRAM. NVWAL's write-ahead log in the PM reduces write operations by using differential logging, which calculates and records only the changing part, not the entire page. Both FASH and FAST use PM for buffer caching and logging. The difference between FASH and FAST is the use of the in-place commit scheme.

Since persistent memory is not yet commercially available, we use Quartz [12], which is a software-based persistent memory emulator. Quartz comprises a kernel module and a user-mode library that emulates PM by injecting software delays per each epoch and throttling the bandwidth of remote DRAM using thermal control registers [17]. Quartz cannot emulate both latency and bandwidth at the same time. Hence, we emulate the latency of PM using Quartz while the bandwidth of PM is set equal to that of DRAM because our experiments are more sensitive to latency than bandwidth.

Quartz [12] cannot emulate the write latency of PM yet, we used Quartz to emulate only the read latency of PM. For emulating PM write latency, we inject an additional delay after each cache line flush instruction, as was used to emulate PM latency in [2, 18]. For a store instruction, we do not insert the delay as the CPU cache can hide it. Quartz runs application threads in a PM-only mode and

DRAM+PM mode. For DRAM+PM mode, Quartz uses two sockets, one is for volatile memory and the other is for persistent memory. Hence, we modify NVWAL code using pmalloc() and pfree() functions of Quartz to allocate virtual persistent memory for write-ahead logs.

The time results of all experiments except the last query processing throughput experiments are measured on database buffer caching and B-tree code without SQL parsing and SQLite bytecode processing. All results reported in this section are the average of 5 runs where for each run we take the average of 100,000 insertions each invoked through an INSERT database transaction statement with randomly generated keys unless otherwise stated.

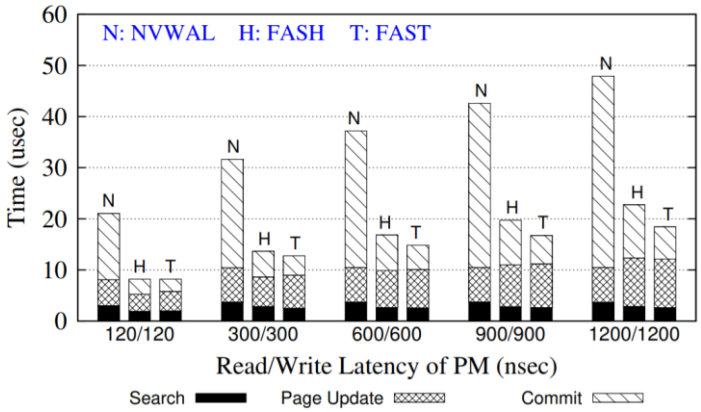


Figure 8. Breakdown of Time Spent for B-tree Insertion
in SQLite as Read/Write Latency of PM is Varied

In this experiment, a 64 bytes record is inserted per transaction, with both read and write latency of PM varying from 300 nsec to 1.2 usec. The leftmost result, 120 nsec, was measured by the latency of DRAM. The insertion time of FASH, FAST and NVWAL are divided into three parts. PM latency is increased 4 times, but insertion time do not increase 4 times. It is because of CPU cache effect. FASH and FAST cannot see the effect of fast DRAM but they are faster than NVWAL and even 2.5~1.6 times faster at PM latency of 1.2 usec.

Insertion time consists of three parts, search time, page update time and commit time. First, the search time is the time to find the leaf page to insert a record in B-tree. Hence, this time is affected only by read latency of PM. Next, the page update time means the time to insert the record into the leaf page found. In the case of FASH and FAST, it includes the time to write an updated slot header in the persistent slot header log. However, the time to flush the log is not included. Finally, commit time refers to the time to flush the redo log to PM and complete the transaction. The commit time of NVWAL includes the time to calculate the dirty part, copy it to the write-ahead log, and then put the commit mark. In FAH, commit time includes flushing the updated slot header to the slot header log,

checkpointing after taking a commit mark, and clearing the slot header. FAST is the same as FASH when using slot header logging scheme, but additionally includes the time for in-place updating of slot header through RTM when only a single page is modified.

In Figure 8, we can see that the page update time of NVWAL is slower than our failure-atomic slotted paging when the latency of PM is same as a DRAM. This is because existing SQLite uses the copy-on-write method to rebalance records when the page split occurs. Since FAST and FASH do not require modification to the page where overflow occurs as shown in section 4.3, the required number of memory write operations is small. However, as the latency of PM increases, the update time of FASN ad FAST becomes longer than that of NVWAL. It is because the overhead of cache line flush and memory fence instructions for the record is increased while updating the record in-place in FASH and FAST. In NVWAL, the page update time is not significantly affected by PM latency because the work is executed only in the volatile buffer cache.

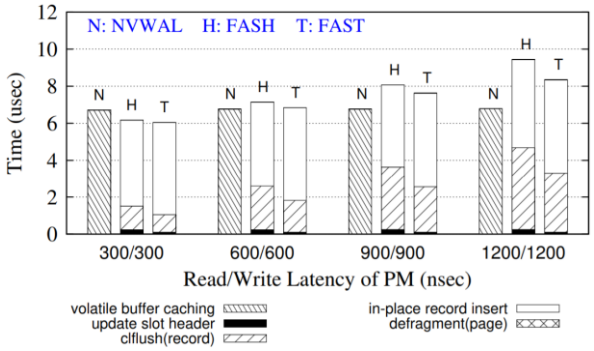


Figure 10. Break of Page Update Time Spent for B-tree Insertion in SQLite as Read/Write Latency of PM is Varied

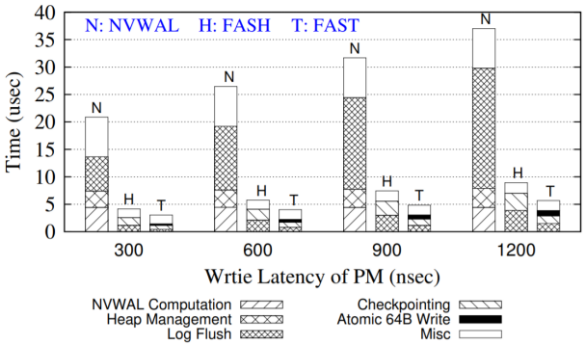


Figure 9. Break of Commit Time Spent for B-tree Insertion in SQLite as Write Latency of PM is Varied

Figure 9 shows a breakdown of page update time of FASH, FAST, and NVWAL. Since the page update time of NVWAL is not affected by the latency of PM, the time of NVWAL does not change even when the PM latency increases. The reason why in-place inserts of records in FASH and FAST (*in-place record insert*) are faster than NVWAL's *volatile buffer caching* is that they use only the record instead of the whole page and the differences of split method. The cache line flush instruction time for the record (*clflush(record)*) increases as the PM latency increases. The *update slot header* filed is time to update the slot header and copy it to the slot header log. However, this filed does not include cache line flush instructions for the slot header log. We can see that *update slot header* time and *defragmentation* time are very small.

Figure 10 shows the result of the breakdown of commit time. In this graph, read latency of PM is fixed at 300 nsec and only write latency of PM is varied because FASH, FAST, and NVWAL mostly

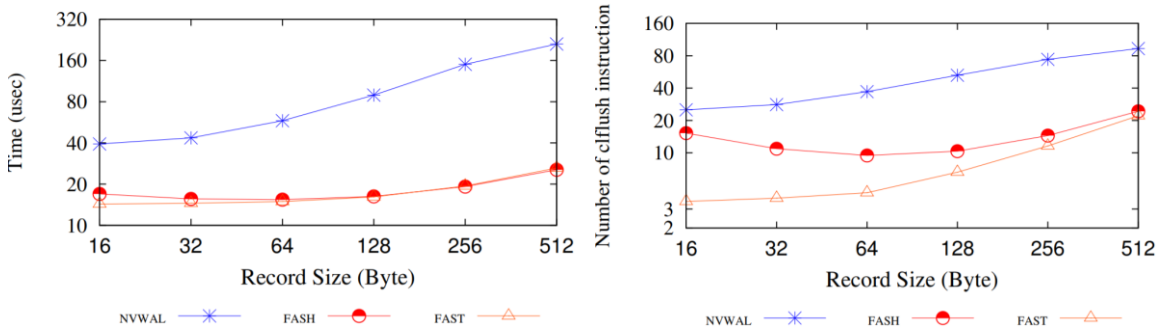
use write operations in the commit phase. The total commit time shows that the time of FAST is up to 6 times faster than NVWAL.

The commit time of NVWAL is considerably slow among three schemes. One of the reasons is that NVWAL includes *computation time* and *heap management time* that are not present in FASH and FAST. Computation time is the time to calculate the differential logging which takes about 4 usec in total commit time. NVWAL also uses a user-level heap manager to manage the PM address space, which takes up about 3 usec.

The *log flush time* is the part where the difference between failure-atomic slotted paging and NVWAL is obvious. It refers to the time at which cache line flush and memory fence instructions are executed to store the WAL frame. NVWAL is significantly slower because the size of WAL frame and WAL frame header calculated by NVWAL is 4 times to 8 times larger than the size of slot header. Even FAST has faster log flush time than FASH because it uses the in-place commit scheme when page split does not occur. In the figure, the atomic 64B write time executed in the in-place commit scheme is also very small.

The part that exists only in FASH and FAST is *checkpointing time*. The checkpointing overhead of FAST is 0.72 usec, which is 49% smaller than FASH's overhead (1.42 usec). It is because checkpointing does not occur when FAST uses in-place commit scheme. Unlike the eager checkpointing of FASH and FAST which is performed for each transaction, NVWAL uses a lazy checkpointing method which is executed periodically, so Figure 10 does not include the checkpointing time of NVWAL.

Finally, the miscellaneous computation portion that FASH, FAST, and NVWAL have in common also has the largest value in NVWAL. The primary cause is the time to build an index for the WAL frame of NVWAL that is not in failure-atomic slotted paging.



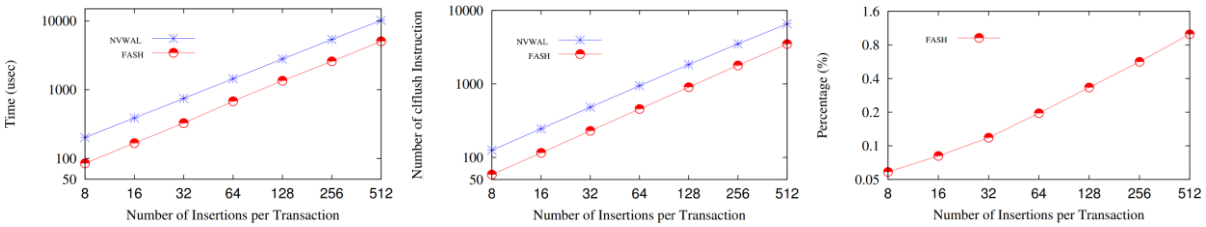
(a) Insertion Time (b) Number of clflush per Insertion

Figure 11. Insertion Time as Record Size is Varied

The experiment in Figure 11 measures the performances of FASH, FAST and NVWAL while inserting records of various sizes. Both read and write latency of PM are fixed at 300 nsec. 11(a) shows the average time for a single transaction to perform insertion query. The larger the size of the record, the greater the performance gap between NVWAL and our failure-atomic slotted paging. FASH and FAST perform an in-place update on the record, but NVWAL copies the record to volatile buffer cache, write-ahead log and the DB file. Hence, as the record grows, NVWAL is more heavily influenced by amplification of redundant memory writes.

11(b) compares the average number of cache line flush instructions per insertion transaction. Most noticeable is that the number of cache line flush instructions of FASH increases as the record size decreases when the size of the record is smaller than 64 bytes. If the record size is small, more records can be inserted on a single page, requiring more cache line flush instructions for larger slot headers. For example, a 1 KB slotted page can contain fifty records of 16 bytes, and the slot header size can be up to 106 bytes (8+50*2). In the case of a 64 bytes record, maximum 15 records can be included in one page, and the slot header is 38 bytes (8+15*2). Since the slot header logging scheme duplicates the slot header, if the record is smaller than 64 bytes, the size of slot header becomes more important than the record that requires a single cache line flush instruction in common.

In the case of FAST, since the in-place commit scheme limits the size of the slot header to a cache line, it calls cache line flush instructions less than FASH even when the size of the record is small. Thus, when the record is smaller than 64 bytes, approximately 3 cache line flush instructions are used on average, one is for the record, another is for the slot header, and the other is for the amortized overhead of page split. The larger the record, the larger the cache line flush overhead. So, the performance gap between FASH and FAST is reduced.



(a) Insertion Time

(b) Average Number of clflush per transaction

(c) Proportion of clflush for Defragmentation

Figure 12. Insertion Performance as Number of Insertion per Transaction is Varied

Figure 12 measures insertion time and the number of cache line flush instructions while the number of 64 bytes record insertion per transaction is varied. In the graphs, x and y-axes are log scale. The

read and write latency of PM is 300 nsec. Since this experiment modifies several pages in a single transaction, FAST cannot be tested.

Figure 12(a) and 12(b) show that the query execution time and the number of cache line flush instructions increase linearly with the number of insertions. When a transaction inserts 8 records, FASH calls average 58 cache line flush instructions and NVWAL calls 125 cache line flush instructions, which is about 2.2 times more than FASH. While the number insertion query per-transaction increases, the possibility of inserting records into the same page increases. If a record is inserted into a split page, FASH and FAST perform defragmentation. Hence, the slot header logging scheme has copy-on-write overhead in this case, which causes more performance degradation than NVWAL. However, even with 512 insertions in a single transaction, FASH calls half of the cache line flush instructions of NVWAL.

Figure 12(c) shows the ratio of cache line flush instructions called by defragmentation among the total number of cache line flush instructions. As we can see from the figure, defragmentation overhead takes about 0.06% for 8 insertions per transaction and it only takes 1% even for 512 insertions per transaction.

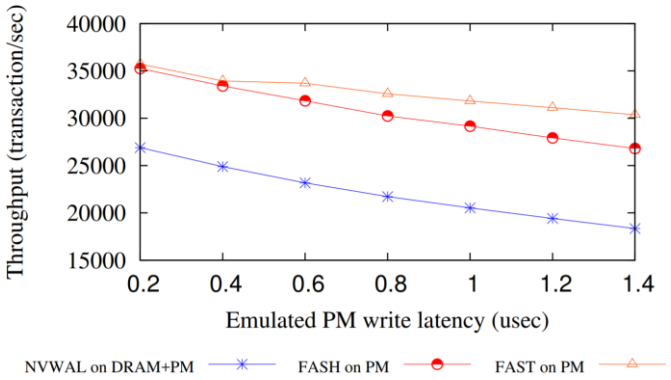


Figure 13. End-to-end Transaction Throughput

Figure 13 considers real word effects. We exploit Mobibench [19] which inserts 1,000 records of 135 bytes. It measures end-to-end throughput including SQL parsing overhead and SQLite bytecode processing overhead. We do experiment with only varying the PM write latency.

When the PM write latency is 200 nsec, NVWAL has an end-to-end throughput of 26,890 transactions/sec. Compared to NVWAL, FASH has 31% (35,251 transactions/sec) and FAST has 33% (35,754 transactions/sec) higher performance. When the PM write latency is 1.4 usec, the throughput of FAST is reduced by 15% (30,365 transactions/sec) compared to the throughput at 200 nsec. It means that database transactions are less sensitive to PM write latency.

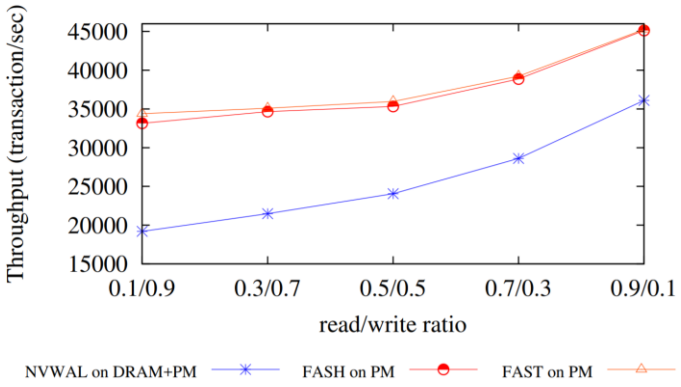


Figure 14. Mixed Workload

FASH and FAST reduce the write traffic significantly by using the persistent buffer caching. However, since PM has higher read latency than DRAM, it cannot receive the effect of the fast read operation of the volatile buffer cache. Figure 14 shows the results of measuring query processing throughput while varying the ratio of search and write transactions. We first created a table with 100,000 records and then write or read 10,000 records of 64 bytes. PM latency is set as 1.2 usec for both read and write.

Since read transaction is faster than write transaction, the throughputs of FASH, FAST and NVWAL increase as the read ratio increases. NVWAL is favorable for the search transaction because it uses volatile buffer cache, but the throughput of FASH and FAST is higher than NVWAL even when the read ratio is 90%. This is because the write transaction has a greater impact on the overall query processing performance [8]. This experiment suggests that PM-only data buffer caching may perform better than hybrid memory system using both PM and DRAM.

VI. Conclusion

Emerging persistent memory is used to address the problem of excessive write operations in the database recovery technique. In this work, we target persistent database buffer cache by replacing DRAM with PM and propose novel failure-atomic slotted paging schemes. The persistent slotted page can write a record without overwriting the committed records and the dirty record is converted to valid record only after the slot header is updated. Our in-place commit scheme updates the slot header atomically using hardware transactional memory and slot header logging scheme stores the slot header into the separate persistent slot header log as redo information. They effectively eliminate unnecessary redundant copies of database pages and minimizes the number of write operations.

We evaluated our proposed failure-atomic slotted paging against NVWAL, which is the state-of-the-art logging method leveraging both DRAM and PM. Our experiments showed that FAST shows optimal performance – only 3 cache line flushes for database transactions that insert just a single record. Even for larger transactions that insert more than one records, FASH reduces the logging overhead to at least 1/4 and up to 1/6 compared to NVWAL. These results imply that PM-only memory architecture may perform faster than hybrid memory architecture.

Reference

[1] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133-146: ACM.

[2] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in Write-Ahead Logging," presented at the Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, Georgia, USA, 2016.

[3] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," presented at the Proceedings of the 11th USENIX conference on File and Storage Technologies, San Jose, CA, 2013.

[4] K. Junghoon, M. Changwoo, and E. Young, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Transactions on Consumer Electronics,* vol. 60, no. 2, pp. 217-224, 2014.

[5] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *Proc. VLDB Endow.,* vol. 8, no. 12, pp. 1454-1465, 2015.

[6] D. Narayanan and O. Hodson, "Whole-system persistence," presented at the Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, London, England, UK, 2012.

[7] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," presented at the Proceedings of the 2013 USENIX conference on Annual Technical Conference, San Jose, CA, 2013.

[8] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split," presented at the Proceedings of the 12th USENIX conference on File and Storage Technologies, Santa Clara, CA, 2014.

[9] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," presented at the Proceedings of the 12th USENIX conference on File and Storage Technologies, Santa Clara, CA, 2014.

[10] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw Hill, 2000.

[11] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, p. 15: ACM.

[12] *Quartz*. Available: https://github.com/HewlettPackard/quartz

[13] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 27, no. 5, pp. 1537-1550, 2016.

[14] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," presented at the Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, Davis, California, 2013.

[15] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, 2014, pp. 580-591: IEEE.

[16] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, p. 26: ACM.

[17] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A Lightweight Performance Emulator for Persistent Memory Software," presented at the Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, 2015.

[18] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," *Proceedings of the VLDB Endowment,* vol. 8, no. 4, pp. 389-400, 2014.

[19] *Mobibench*. Available: https://github.com/ESOS-Lab/Mobibench